# DETECTING FORGED BANKNOTES

LOUIS DE THANHOFFER DE VOLCSEY

*september 2017*

## Contents

# 0 Introduction

## 0.1 Overview

In this project we design a neural net capable of reliably detecting forged banknotes.
Our algorithm relies on data that was collected by scanning notes which were then processed statistically. In doing this, one is able to distill some of the key features that describe the inherent mathematical information inherent to each image.
Our approach can be subdivided in two major phases:

1. **reengineering the data**. Based on the famous machine learning folklore:

   *machine learning starts with feature engineering,*

   we perform an extensive analysis of the data through various statistical tools before fitting it to a neural net. Some of the techniques we use include: detecting and removing outliers, describing the various relations between different features, visualizing the data in 3d and 2d as well as analyzing to what extend the data can be separated by a decision boundary.[1]

2. **Designing a neural network** Once the data has been analyzed, we use the `keras` Python library to implement a neural net. To this end, we initially consider a rather simple model and analyze its performance through the use of learning curves. These curves give us not only an insight into how well the neural net can handle the problem, but they allow us to understand how to make modifications in order to improve on the design. Our final model will be capable of accurately detecting forged banknotes **99%** of the time all the while retaining a simple and transparent architecture.

## 0.2 Background to the problem

Known colloquially as one of the *worlds second oldest profession* [sopon], the art of counterfeiting legal tender dates back to the invention of money itself. From forgers reproducing gold coins in Roman times by using cheaper base metals [wikit] to the use of Mulberry trees [wikit] in order to imitate banknotes in 13th century China, forgers and governments have always played a fiendish cat-and-mouse game that makes for a fascinating history. Counterfeiting techniques were even used as a form of warfare in the U.S. Revolutionary war by the British in order to devaluate the newly created U.S. dollar [wikit].
The problem of counterfeiting however, is not just one of historical importance. In fact, it is estimated that this issue produces an annual increase in inflation that results in a loss of purchasing power of over 250 billion dollar annually to the U.S. economy [usf36].
In modern times, the techniques to detect forgery have grown very subtle as anti-counterfeiting measures enter the 21st century (see [wikit]), ranging from the use of complicated printing techniques to the inclusion of watermarks. The main paradigm behind these techniques has however remained unchanged throughout history: at its core, a note is fitted with a set of attributes and run through a series of tests to determine if those attributes are indeed present .
This project aims to use state-of-the art machine learning techniques to detect forgery in an

[1] we refer the reader to §efconc1 for a more technical overview account of these techniques

entirely different way. Instead, using mathematical analysis, the banknote will inherently be described in waveform. We will subsequently use the technology of neural nets to devise a formula which uses some inherent statistical features of this wave allowing us to detect forgery.

# 1 Analyzing the dataset

The dataset in question was extracted from the UC Irvine machine learning repository [2]. It serves as the basis for a regression analysis performed by Gillich and Lowesh in [GLon]. In loc. sit. the authors scanned a set of 1372 banknotes into 400× 400 pixels and applied a *Wavelet transform* to each image. The wavelet transform is a mathematical tools which finds its origins in the theory of Fourier transforms and allows one to encode image data very efficiently. For our purposes it will be sufficient to know that it compresses an image by extracting coefficients according to an underlying probability distribution. The features of the dataset in question consist of a description of 4 statistical properties of this distribution:

1. the variance
2. the skewness
3. the kurtosis
4. the Shannon entropy

Additionaly, to each feature a binary label is associated indicating whether or notethe note is real or counterfeit (0=real, 1=fake). In the dataset, the first 762 notes are real, whereas the last 610 are counterfeit. By way of example, the 350'th datapoint is the 5-tuple:

```
[-1.5768,10.843,2.5462,-2.9362,0]
```

## 1.1 a First exploration

We begin by describing the basic statistical properties of the data. These are listed below as well as displayed graphically in the form of a boxplot:

| statistic | variance | skewness | curtosis | entropy |
|---|---|---|---|---|
| mean | 0.43 | 1.92 | 1.40 | -1.19 |
| std | 2.84 | 5.87 | 4.31 | 2.10 |
| min | -7.042 | -13.77 | -5.29 | -8.55 |
| 25% | -1.77 | -1.71 | -1.57 | -2.41 |
| 50% | 0.50 | 2.32 | 0.62 | -0.59 |
| 75% | 2.82 | 6.81 | 3.18 | 0.39 |
| max | 6.82 | 12.95 | 17.93 | 2.45 |

**Table 1**: describing the data

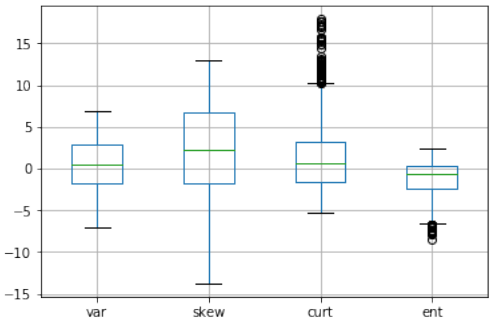



**Figure 1**: the boxplot of the data

We see that all 4 features have rather similar statistical properties. As a result we opt to not renormalize the data, preferring to keep transparency
A closer look at the boxplot does yield an interesting observation: it seems that a significant

[2] https://archive.ics.uci.edu/ml/datasets/banknote+authentication

number of datapoints have a kurtosis lying above the boxplot whiskers. These should be considered outliers. Similarly, the entropy feature, seems to present a significant number of outliersas well. After counting these, we removed a total of **65** datapoints (equivalent to **4.3%** of the total data). The new dataset now consist of **1305** entries and presents no outliers.

## 1.2 the Correlation between features

The next step in our data exploration, consists of investigating the possible correlations between mutual features. This is an important step as correlated features morally correspond to redundant information, which may cause overfitting as we build a model later on.
The Pearson correlation coefficients for each choice of feature pair together with their respective scatterplots are listed below. [3]

| feature | var | skew | curt | ent |
|---|---|---|---|---|
| var | 1 | 0.26 | -0.38 | 0.27 |
| skew | 0.26 | 1 | -0.78 | -0.52 |
| curt | -0.38 | -0.78 | 1 | 0.32 |
| ent | 0.27 | -0.52 | 0.32 | 1 |

**Table 2**: the Pearson correlation matrix

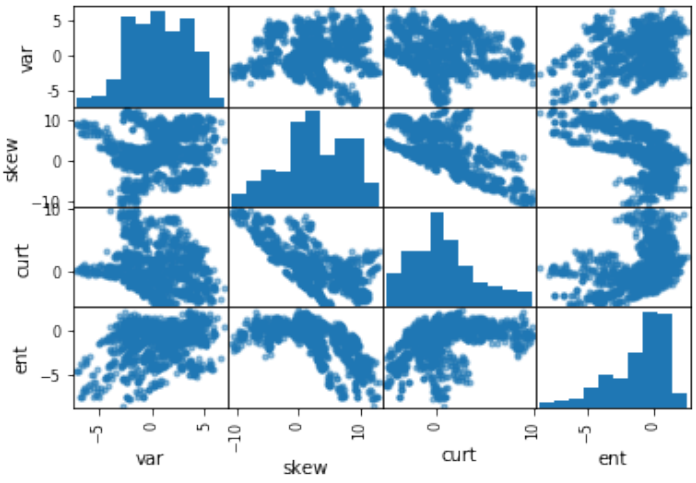



**Figure 2**: the scatterplot of the features

Both the Pearson matrix and the scatterplot indicate that exactly two features are significantly correlated: kurtosis and skewness.
Comparing these features, yields a Pearson correlation of 78% in absolute value. Additionally, the scatterplot shows that linear trend seems to form around the $y = -x$ axis between those features. We choose however to keep those features as removing either would come at the cost of significantly reducing the dataspace, which in turn could impact the predictive capabilities of the final model. Instead, since we intend to analyze the model through learning curves later on we will be able to detect overfitting as a difference in behavior between training and testing accuracy and will adapt the model accordingly if necessary.

## 1.3 Reducing the dimension

The third step in our analysis consists of reducing the dimension of the features to $3d$ and $2d$. The tool of choice for such problems is a *principal component analysis*, This method determines an optimal subspace by maximizing variance (or equivalently minimizing mean squared error), subsequently exhibits an orthonormal basis for this subspace. From there, the coordinates for any projected vector with respect to this basis are computed and returned. For the reader's benefit. A pseudo-algorithm of this procedure in sklearn is given below:

[3] we note that in the case where two identical features are chosen, their histogram is displayed

**Algorithm 1**: reduce dimensionality using PCA

```
function reduce (features, d);
  Input  : a set of features together with the desired final dimension
  Output: a set of d-dimensional features, together with the total variance of each dimension
pca=PCA(n_features=d)
pca.fit(features)
print(pca.explained_variance_ratio_)
reduced_data=pca.transform(features)
return reduced_data
```

The total variance for each principal component after applying a PCA transformation is listed below:

| component | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| expl. var. | 76% | 14% | 6% | 3% |

## 1.4 Plotting the data

Reducing the dimensions of the features allows us to observe how the binary labels are distributed. In the figures below, the features were reduced using the reduce function and then plotted. The red labels correspond to real banknotes whereas the green labels represent forgeries.

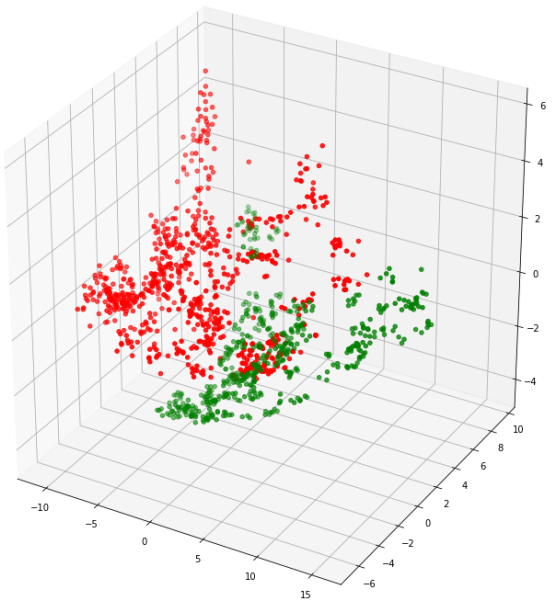


**Table 3**: the 3d reduced data

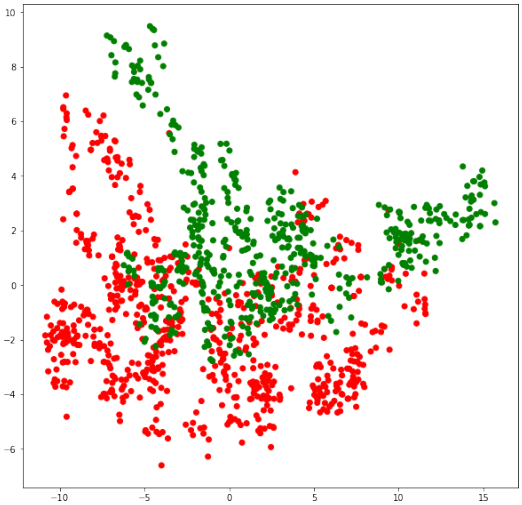


**Figure 3**: the 2d reduced data

These plots indicate that the reduced data is not linearly separable. What's more, a conclusive decision boundary is difficult to discern. This hypothesis can be confirmed numerically through the use of a support vector machine. This clustering tool maximizes the *marginal distance* between two classes of datapoints in order to compute a hyperplane which separates them optimally. In

particular, should the classes be linearly separable, the accuracy of the algorithm would be 100%. Below, we first apply the reduce function described in **Algorithm 1** and then fit a support vector machine and compute its accuracy -which in this case simply the relative number of correctly classified points:

**Algorithm 2**: compute the accuracy of a separating hyperplane after reducing dimensions

```
function lin _sep_acc (features, m);
Input  : a set of features
Output: the accuracy of a separating hyperplane after reducing dimensions up to m
d=features.shape[1]
for i in range(m,d+1) do
    reduced_features=reduce(features,i)
    clf=SVC()
    clf.fit(reduced_features,labels)
    predicted_labels=clf.predict(reduced_features)
    accuracy_score(predicted_labels,labels)
end
```

The results of this algorithm are listed in the table below:

| dimension | 4 | 3 | 2 |
|---|---|---|---|
| accuracy | 81% | 78% | 59% |

This indeed confirms our hypothesis that the data cannot be linearly separated, even after reducing dimensions.
It is worth noting that one could alternatively implement a support vector machine with a more sophisticated kernel that separates the data using more general algebraic varieties for instance. It turns out that -for example- allowing for a cubic kernel does not improve the accuracy as the table below shows:

| dimension | 4 | 3 | 2 |
|---|---|---|---|
| accuracy | 81% | 78% | 42% |

## 1.5 Conclusions

Our analysis of the data has let us to the following conclusions:

- The dataset -consisting of 4 features with binary labels- is roughly equidistributed. As such it is not necessary to perform any rescaling operation.
- A closer look at the boxplot does show outliers however. After removing these, 95.7% of the mass is retained.
- Within the data, two features are heavily correlated: computing the Pearson coefficient of kurtosis vs. skewness results in a |72%| correlation. This could be a possible explanation of overfitting later on.
- Applying a principal component analysis allows us to reduce the data to 3 or 2 dimensions and to conclude that the distribution of the labels is rather heterogeneous. In particular the data does not present an obvious decision boundary.

- Fitting a support vector machine (possibly after reducing dimensions) supports this evidence as this results in misclassification about 20% of the time.

# 2 Building a neural net

The analysis in the previous section led us to the conclusion that the dataset is potentially vulnerable to overfitting and that a simple decision boundary does not exist. In light of these facts, we opt to use a neural network to classify the data, as these are known to be both robust enough to handle more complicated datasets and transparent enough in their design.

## 2.1 Neural nets in Keras

To implement a neural net, we will use the `keras` Python library using a `TensorFlow` backend. This library (built specifically for neural nets) has become one of the gold standards in the industry due to its ease of use and its modularity. In fact, the programming paradigm in `keras` is remarkably streamlined: in short, a network is built in two steps:

- **step 1** the user designs the net layer by layer. For each layer they can easily specify a chosen number of input nodes, an activation function or implement regularization functionalities such as dropout.
- **step 2** The design is subsequently compiled and fitted to a given training/testing set using a gradient descent method. One pleasant feature of `keras` is that the parameters of the gradient descent method can easily be tweaked, which results in a highly customizable optimizer.

As was mentioned in the introduction, our approach will be to start with a rather coarse neural net and refine the design as we make conclusions based off the resulting learning curves. To streamline the code, we chose to implement **step 2** as a separate function. To this end we first split the data into a training (2/3) and testing (1/3) set.[4].
The algorithm below fits a neural net using this training/testing set according to a prescribed model with a specified number of epochs and choice of optimizer:

**Algorithm 3**: compile and fit a neural net

```
1 comp_fit(model, epochs, sgd);
  Input  : a keras neural net, a number of epochs, a gradient descent optimizer
  Output: a fitted model
2 model.compile( optimizer=sgd, loss='categorical_crossentropy', metrics=['accuracy'])
3 fm=model.fit(x_train, y_train, validation_data=(x_test, y_test), epochs=epochs,batch_size=10)
4 return fm
```

As part of our preparatory work, a function learning_curves was implemented as well. This function displays both the accuracy and the value of the loss function for training and testing sets after each individual epoch.

[4] As a technical note, we warn the reader that numpy data needs to be transformed slightly in order to make it compatible with keras: the features need to be converted into matrices and the labels need to be one-hot-encoded

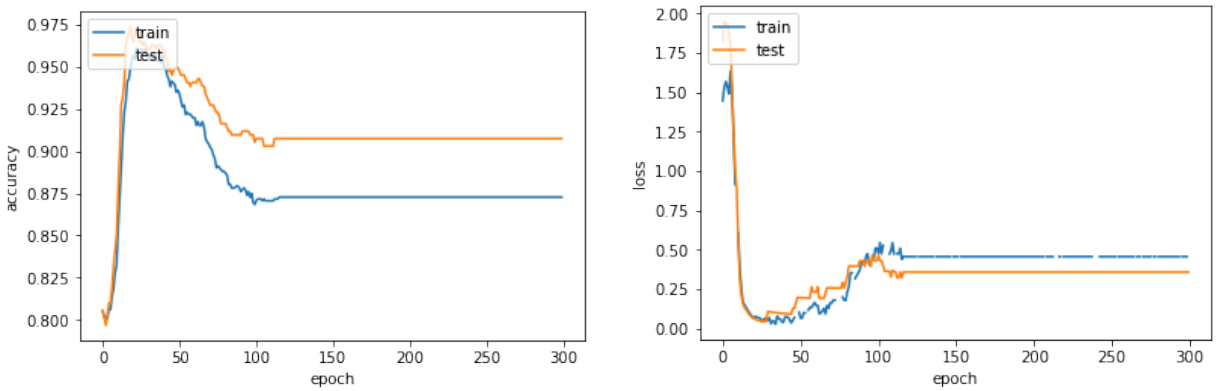
## 2.2 Finding the right model

**2.2.1. A coarse model** We begin our search for the optimal model with a standard implementation of logistic regression. This neural net has no hidden layers and uses a sigmoid activation function. For the benefit of the reader we describe an implementation of this neural net using the standard `rmsprop` optimizer with 300 epochs:

**Algorithm 4**: logistic regression in `keras`

```
log_model();
Output: a a fitted model for logistic regression together with the learning curves
log_model=Sequential()
log_model.add(Dense(2,input_dim=4,activation='sigmoid'))
history=comp_fit(logmodel,'rmsprop',300)
learning_curves(history)
```

The learning curves for this model are displayed below




Through these learning curves, we are able to draw a number of conclusions:

1. the final accuracy on the training set is 86% . This does not improve significantly on the 81% accuracy of the support vector machine discussed in §.1.4, or in other words, the model is underfitted as well. This points to the fact that our coarse model is too basic to handle the dataset. To remedy this, we will investigate how the results change as we add hidden layers to the net in §.2.2.3 .

2. The learning accuracy on the testing set is 90&, implying that the algorithm will still misclassify one out ten new banknotes.[5]

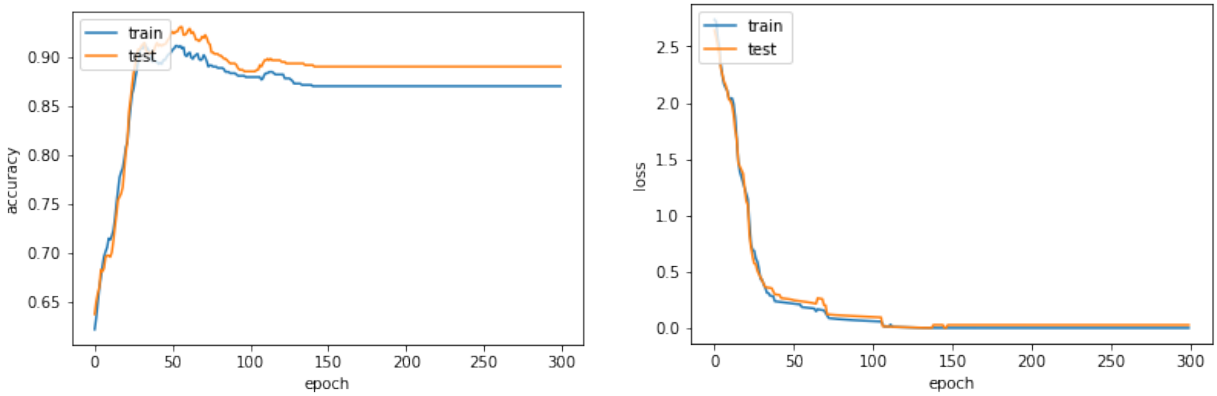
3. We also see a nonnegligable difference between both training and testing accuracies. In particular, the testing set has a higher accuracy than the training set, which doesn't typically happen. Since our dataset is comparatively small, this could be a result of how the the data was split. To account for this, we altered the seed for the pseudorandom generator used to split the data.

4. Both in the case of training and testing, the loss stagnates at around 0.4/0.5 after 110 epochs. As a result, the accuracy stagnates as well since the algorithm is no longer optimizing itself.

[5] we note that a comparison with the SVM cannot be made here as it was not tested on new examples. Instead we took the complete dataset into account to conclude that the data could not be separated linearly

Geometrically speaking, the optimizer has stumbled upon a value that is a local minimum for the cost function. This value is arguably still rather high as the function has only dropped around 80%. As part of our exploration, we will tweak the parameters of the optimizer to see if this result in a more efficient gradient descent method in the paragraph below:

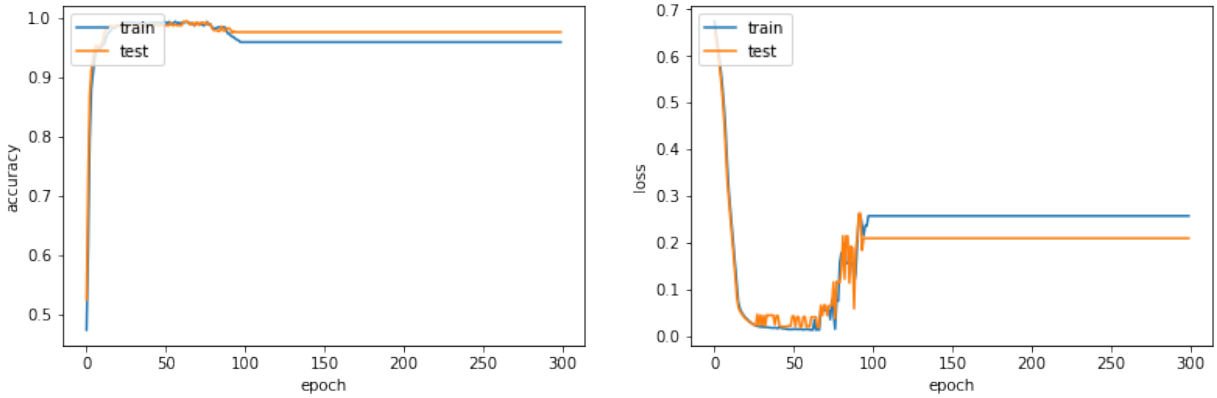
**2.2.2. Optimizing the optimizer** Tweaking the parameters of the optimizer in the previous model can be done mutatis mutandis by replacing the default `rmsprop` in line 4 of **algorithm 4** by a custom one using the command `SGD.optimizers()` and specifying the chosen parameters. The analysis done in §2.2.2.1 points to the fact that -for example- decreasing the learning rate could improve the accuracy as it would give the optimizer more freedom to "explore" before it finds a local minimum. After a few trials with different learning rates, we found that the `Adam` optimizer built into `keras` -whose learning rate is one tenth of `rmsprop` indeed showed some improvment. The learning curves for this model are displayed below:




These curves confirm our assumption:

1. After 150 epochs, the cost function has decreased drastically to around 0.1. resulting in significantly better optimization.
2. The results on accuracy however have remained unchanged, implying that we will need to alter the design of the neural net.

**2.2.3. Adding a hidden layer** The next step in our model design is to add a hidden layer of neurons. One major factor to take into consideration here is the number of inputs. This is a delicate exercise in over/underfitting: too many nodes will result in a model that overfits the data, whereas too few nodes will not result in any significant increase in accuracy.
After a few trials, we found that increasing the number of nodes by more than 4 does not results in any improvement. Adding a layer of 4 hidden neurons with a sigmoid activation, results in a training accuracy of 95% and a testing accuracy of 97% as the learning curves below show:




**2.2.4. Implementing Dropout** Since the analysis summarized in §. 1.5 showed that the data is prone to overfitting, as a final step we investigate if adding a Dropout layer improves our model. This drastic technique is typically used to prevent overfitting in deep nets trained on large datasets and consists of deactivating neurons at random (according to a probability rate) in order to force the model to keep learning. This functionality can be implemented easily in `keras` through the command `model.add(Dropout())`. For the benefit of the reader, we include the psuedocode for our final algorithm:

**Algorithm 5**: logistic regression in `keras`

model();

**Output**: a fitted neural net with an Adam optimizer, hidden layer and Dropout

model=Sequential()

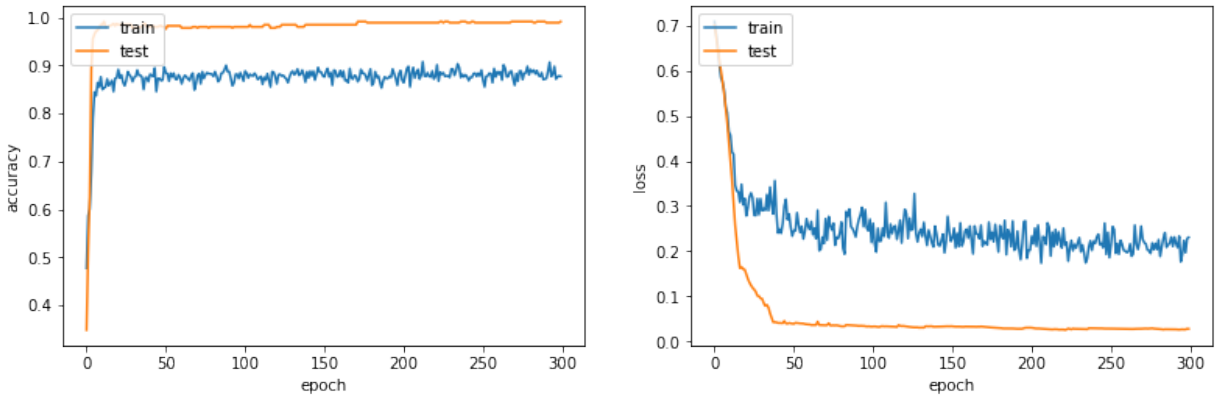
model.add(Dense(4,input_dim=4,activation='sigmoid'))

model.add(Dropout(0.5))

model.add(Dense(2,input_dim=4,activation='sigmoid')) history=comp_fit(logmodel,'Adam',300)

learning_curves(history)

After adding a Dropout layer between the input and hidden layer, the learning curves were as follows:

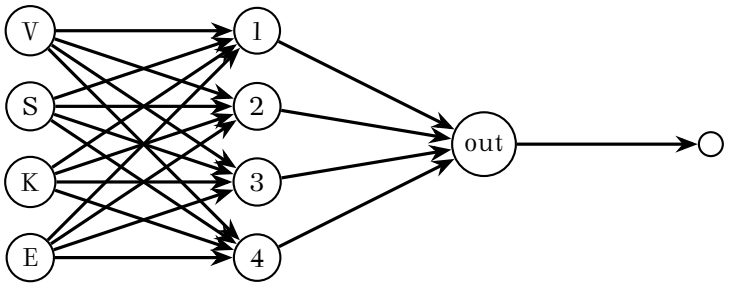



The learning curves for the training set are more stochastic in nature, due to the randomness inherent to the Dropout method. We can see that the method did produce favorable results as the accuracy on the testing set is 99%

# 3 Summary

We began our investigation of the problem by preprocessing the data through the use of various statistical- and machine learning techniques which allowed us to gain insight into the data's nature.
Along the way, we descovered that the features were equidistributed,; we detected and removed the necessary outliers; showed that precisely two features were significantly correlated; reduced the dimension to gain visual insight and determined that a simply clustering technique would not be be suitable for this problem.

In the second stage of our research, we built a neural net to classify the data using `keras`. After implementing a first approximate model based off logistic regression, the learning curves showed that adapting the optimizer would yield better results. After lowering the learning rate, we obtained a model that indeed showed a lower cost, but did not improve on accuracy. We then added a second hidden layer with 4 input nodes to the net, based off trial and error. Finally, based on the idea that the data could be overfitted, we added a dropout layer, which resulted in a model that has a 99% accuracy:




In conclusion, given a scanned banknote, we believe that this model is capable of accurately detecting foul play given the statistical features of an image.

# References

[GLon] E. Gillich and V. Lohweg. Banknote authentication. *Conference Proceedings*, BVau(2010), `https://www.researchgate.net/publication/266673146_Banknote_Authentication`.

[sopon] Wikipedia:worlds second oldest profession. `https://en.wikipedia.org/wiki/World%27s_second_oldest_profession`.

[usf36] Counterfeiting costs us businesses. `http://www.ipwatchdog.com/2010/08/30/counterfeiting-costs-us-businesses/id=12336/`.

[wikit] wikipedia: counterfeiting. `en.wikipedia.org/wiki/Counterfeit`.